The Thesis Committee for Charles Austin Manion
Certifies that this is the approved version of the following thesis:

Assembly Sequencing Through Graph Reasoning: Graph Grammar Rules for Assembly Planning

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Matthew Campbell

Richard Crawford

Assembly Sequencing Through Graph Reasoning: Graph Grammar Rules for Assembly Planning

by

Charles Austin Manion, B.S.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2013

Abstract

Assembly Sequencing Through Graph Reasoning: Graph Grammar Rules for Assembly Planning

Charles Austin Manion, MSE

The University of Texas at Austin, 2013

Supervisor: Matthew I. Campbell

Assembly planning is difficult and tedious, but is necessary for complex products. This thesis presents a novel approach to automating assembly planning utilizing graph grammars. Computational geometric reasoning is used to produce a label rich graph from a CAD model. This graph is then modified by graph grammar rules to produce candidate assembly sequences which are run in conjunction with a tree search algorithm. An evaluation system then evaluates partial assembly sequences, which are used by the tree-search to find near-optimal assembly sequences.

Table of Contents

List of Figures

Introduction

Design for Manufacture and Assembly(DFMA) is an approach to designing products such that manufacturing and assembly costs are minimized. Given that assembly often makes up at least 50 percent of the total cost of a product [1], employing DFMA can reduce product cost. DFMA can also decrease product development time by reducing the need for redesign at the manufacturing phase. However, DFMA is not carried out for many products due to the fact that DFMA can be time-consuming and tedious to perform.

Current DFMA methods require a great deal of input from the designer. Most DFMA methods require that the designer determine what assembly operations must be carried out and must estimate their difficulty. For large assemblies, this can be arduous and it is not guaranteed that that designer will find the optimal sequence of assembly operations to be performed. This makes it difficult for the designer to iterate through different design processes to optimize their product for assembly.

A better means of performing DFMA would be to develop an automated process to find a number of the candidate assembly sequences and present the associated costs and times to the designer. Finding the assembly costs allows the designer to find which parts consistently cost the most to assemble, providing the designer with an opportunity for redesign. Finding the best assembly sequence also makes it easier to go directly into manufacturing. Additionally, one could optimize the assembly sequence for production within a standardized production environment to cut down on manufacturing costs and increase manufacturing flexibility. Current assembly sequencing systems are far from automated and require a great deal of input from the user. This thesis provides work to address this problem.

Problem Formulation

The assembly planning problem is the problem of finding the optimal sequence for product assembly. "Optimal" in this context means the sequence that uses the least amount of time, costs the least, and uses the least amount of tooling. The assembly planning problem can be as simple as finding the sequence from precedence constraints or as complex as planning for low-level tool motion. Given an assembly with all parts in their final configurations, the problem statement is to find the best assembly sequence to move all parts into place, without them interfering with any other part, to get to their final positions. Assembly sequence planning can also be used to estimate the cost of an assembly.

For this thesis we assume only single translation of parts, meaning parts can only move in a straight line path to their final position, that the assembly is free floating, and that the assembly remains in one constant orientation. This has been shown to encompass a great number of real world initial assemblies, but it is likely to be insufficient in cases of assembly and disassembly for maintenance (i.e. replacing a single component in a completed assembly). [2] Given that this thesis is concerned only with initial assembly this should not be a problem. This thesis assumes that all parts are rigid bodies, although future efforts will include flexible components.

Prior Art

Jiménez [3] gives a comprehensive overview of assembly sequencing. At its very simplest, assembly sequence planning can be carried out by asking the user a series of questions about what parts must come before other parts. [4] This has the disadvantage of requiring the user to answer a large number of questions making it impractical for assemblies containing large numbers of parts. Mello and Sanderson [5] [6] [7]developed the And-Or graph approach to representing assembly sequences. In this approach, nodes represent subassemblies and directed hyperarcs (an arc that connects more than two nodes) represent assembly operations to construct the 'from' node subassembly from the 'to' nodes subassemblies. An example on an And-Or graph is shown in Figure 1. This allows assembly precedence information, or the fact that a part must come before another part, to be represented. In their approach, assembly operations are represented by

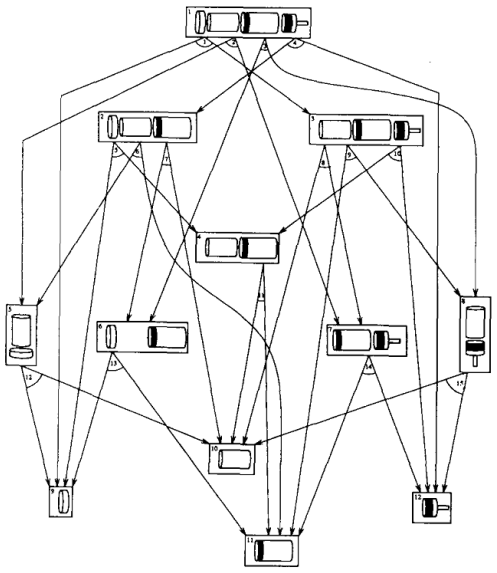


Figure 1: And-Or graph for a simple assembly **[5]**

ANDing and ORing parts together to form subassemblies. One finds the optimal

assembly sequence by searching the And-Or graph for assembly sequences that minimize a given cost function. This approach has the disadvantage that the And-Or graph size grows exponentially with the number of parts in the assembly, making it impractical for assemblies containing more than a few parts. The And-Or graph also requires a great deal of computational effort to generate.

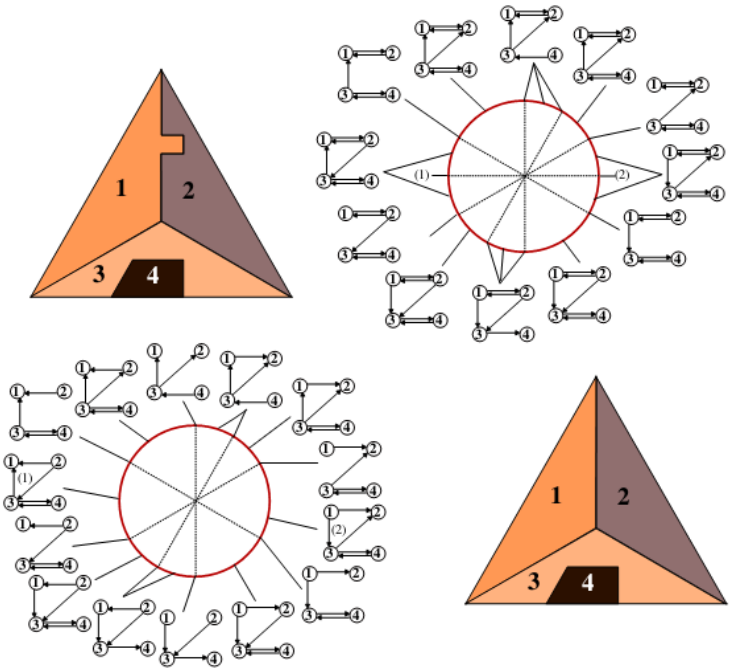


Figure 2: Non-Directional Blocking Graphs and their associated Directional Blocking Graphs[3]

The directional blocking graph (DBG) approach represents part constraint relationships for a given global direction as a directed graph. In the DBG approach, parts are represented as nodes in a graph and directed arcs are placed between parts that block each other in a given direction. For example, if part A blocks part B in the chosen direction, then a directed arc goes from B to A. Collections of parts that can be feasibly removed in the directional blocking graph are given by the strongly connected components of the graph. Strongly connected components are sets of nodes that have a

path that connects all nodes in the set. The existence of a path between components implies that the components block each other and cannot be feasibly separated. Given that strongly connected components do not share a path between each other, by the definition of a strongly connected component, then it can be shown that strongly connected components represent removable subassemblies.

Wilson and Latombe [8] expanded upon the DBG approach with the Non-Directional Blocking Graph approach. A NDBG is a partitioned sphere, or hypersphere, of possible motions divided up into cells of uniform DBGs. An example of this for a 2d scenario is shown in Figure 2. This approach was used in the Archimedes 2 assembly planning system [2], which appears to be the most capable assembly planning system. However, there are few published articles on the approach, no usable computer code has been made available, and a vast amount of human preprocessing was required.

Another approach to assembly planning is to use path planning to find collision-free paths, through which parts can be removed from the assembly. The Rapidly exploring Random Tree(RRT) is one such motion planning approach that has been used in assembly planning systems. [9] [10] This approach works by checking random displacements of a part for feasibility with respect to previous feasible displacements. This approach has the advantage that motion planning is carried out simultaneously with assembly sequencing. This motion planning information can be used to illustrate part assembly operations or as an input to a robotic assembly workcell. The path planning approach is also well suited to finding assembly sequences for assemblies that require complex part motion to assemble. However, this approach has the disadvantage that it tends to be very slow, as a large amount of expensive collision detection must be carried out. The approach is also incapable of planning with subassemblies, as it requires that only single parts be assembled/disassembled in a single step.

Various other methods have been used for assembly planning, including neural networks [11] [12] [13], petri nets [14], genetic algorithms [15] [16] [17] [18], and ant colony optimization. [19] However, these methods are not guaranteed to find an optimal solution.

Geometric Reasoning

To do assembly planning, one must first extract constraint information from the CAD model of the assembly. It is necessary to determine the directions in which components can move or obstruct movement with respect to each other. There are two types of constraints, contact constraints and non-contact constraints. Contact constraints are constraints induced by the direct contact of two parts, while non-contact constraints are constraints that are formed when parts are not in direct contact. For example, the walls of a box constrain the movement of an object inside it, but they do not necessarily need to be in contact to constrain the object. It is relatively simple to derive contact constraints, based off of the position of contact faces one can infer directions a component is free to move with respect to another component.

It is known that certain components such as nuts, bolts, and other standard components will almost always have the same directions of freedom. For example, a bolt will always connect to a threaded component and the bolt can only be assembled onto the axis of said threaded component. These can be tagged by the designer to simplify the constraint extraction process.

Once the geometric reasoning stage is complete, the constraint information is translated to a label-rich graph. Parts are represented as nodes in the graph and connections are represented as arcs. These arcs contain connection type labels and a list of variables representing free directions.

CLASH DETECTION

In order to determine the connection relationship between two components, one performs collision detection to find clashes between components. This could be done by testing for collisions between every single component. However, this would be inefficient, especially for large complicated systems with hundreds, or thousands, of parts. One way to increase the efficiency of the collision detection is to have several phases of varying resolution to avoid performing in-depth collision detection on components that are not in close proximity.

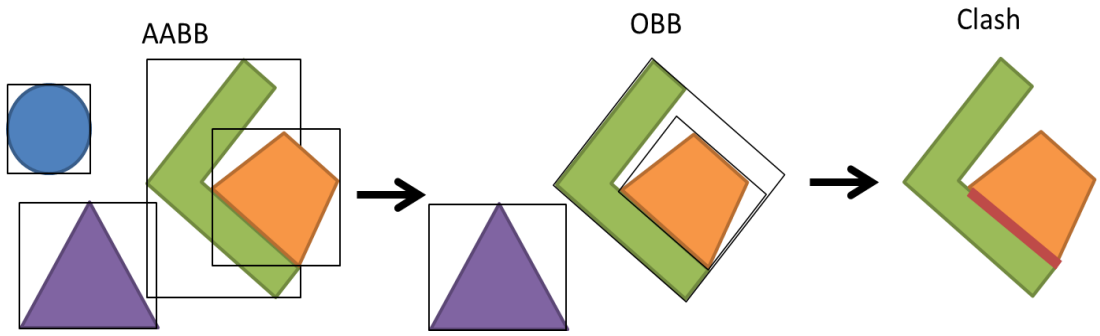


Figure 3: Illustration of Clash Detection Process

For the low-resolution collision detection phases, we use Axis Aligned Bounding Boxes (AABB) and Oriented Bounding Boxes (OBB).(See Figure 3) An axis aligned bounding box is the smallest rectangular box that can contain a part that is oriented with the global coordinate system. As this bounding box is faster to compute, it is used in the first phase of collision detection. AABB's which do not intersect contain parts that can be eliminated from further clash analysis. In the second phase, we use Oriented Bounding Boxes, or the smallest possible rectangular box that a shape will fit into. As this is more expensive to compute, it is done in the second phase. In the final phase of

the collision detection we check the actual Boundary Representation (B-Rep) models for collision. This B-Rep collision detection mode uses the original solid geometry and a CAD kernel [20] to detect an exact collision and determine contact types accurately.

To detect collisions between both OBB and AABB, we use the Separation of Axis Theorem (SAT). SAT is a method of performing collision detection between convex shapes. [21] [22] [23] As SAT encounters problems with shapes that are concave, it can only be used with convex shapes. Given that both the OBB and the AABB are convex; when no collision is detected, there is absolutely no collision between the parts. However, if a collision is detected, full collision detection must be performed using the boundary representation (B-rep) of the parts, as the parts could have concavities. Face to face contacts are only considered, as mechanical assemblies usually contain a large number of them as compared to vertex-to-vertex and edge-to-edge contacts (or any permutation of edge, face, and vertex).

INFERRING NON-CONTACT CONSTRAINTS

In CAD models, there often exist gaps between components that constrain each other due to the way that assemblies are defined in the CAD system. For example, a bolt may 'hover' inside its respective bolt hole due to the necessary tolerancing issues. Fortunately, there will be both axis-aligned bounding box and oriented bounding box collisions, but there will not be any B-rep collisions. To handle this, we perform an additional two tests. We check if the OBB of the first part collides with the B-rep of the second part and perform the same test, but with part representation switched. If both of these tests return true, then we are reasonably certain that a constraint exists between the two parts, as this condition is most likely to occur when a part exists inside another part. To extract the actual constraint, we attempt to find a common cylindrical axis between

the two parts. If there is a circular edge or cylindrical feature, there may be a rotational degree of freedom between the two parts. In the case of cylinder inside a circular hole feature, both the cylinder and the hole should share a common concentric axis. Due to the geometry, motion of both parts should be limited to this axis.

As shown in Figure 4, this axis is found by extracting all circular edges with central angles greater than 180 degrees from both parts. Next, one edge is chosen from one part and a vector is traced from the center of the edge to the center of the edge of the other part. Then this vector is checked against the normal vectors of both edges. If the vector is found to be parallel with both circular edges, then the circular axis has been found. This method could potentially be extended to finding rectilinear non-contact constraints by using rectangles in place of circles.

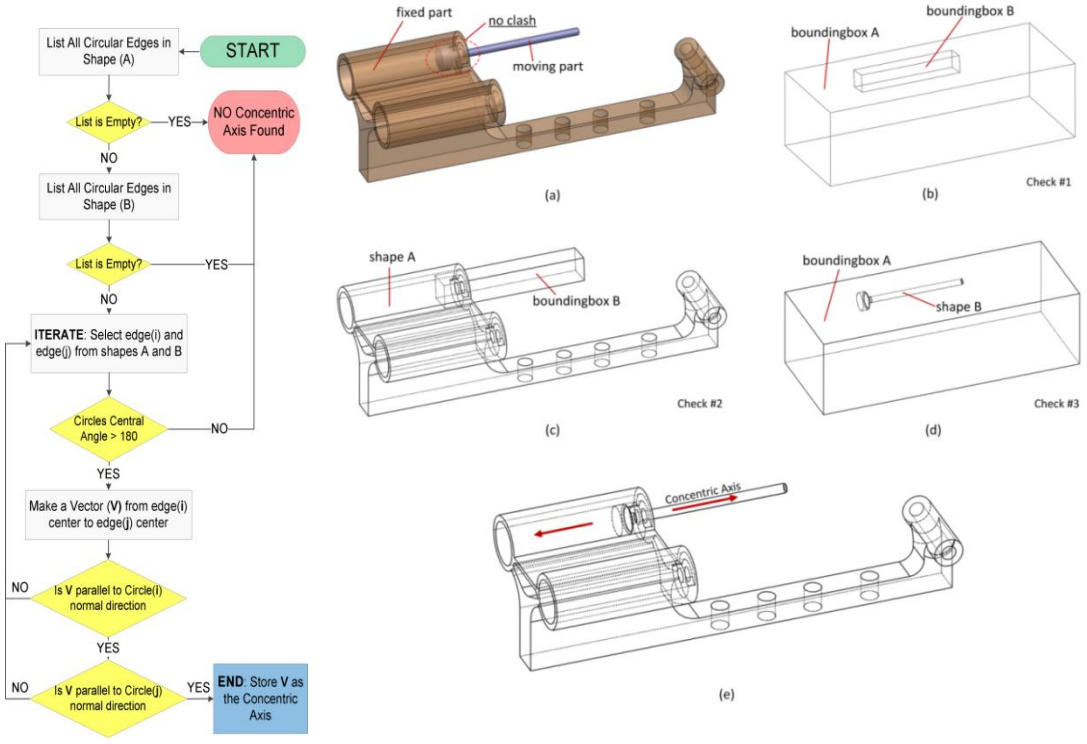


Figure 4: Finding Non-contacting constraints **[25]**

INFERRING CONTACT CONSTRAINTS

The identification of free directions and other constraints is essential to assembly planning. When defining constraints, it is necessary to specify a part that remains fixed with respect to another part. One part is designated as the moving part and another is designated as the fixed part. Determination of fixed and moving parts is determined by the size of the oriented bounding box, the smaller part is chosen as the moving part. This choice is completely arbitrary, and is mainly needed for book-keeping purposes. A reversal of fixed and free parts simply results in a reversal of free-directions. This is represented in the graph with a directed arc going from the free part to the fixed part.

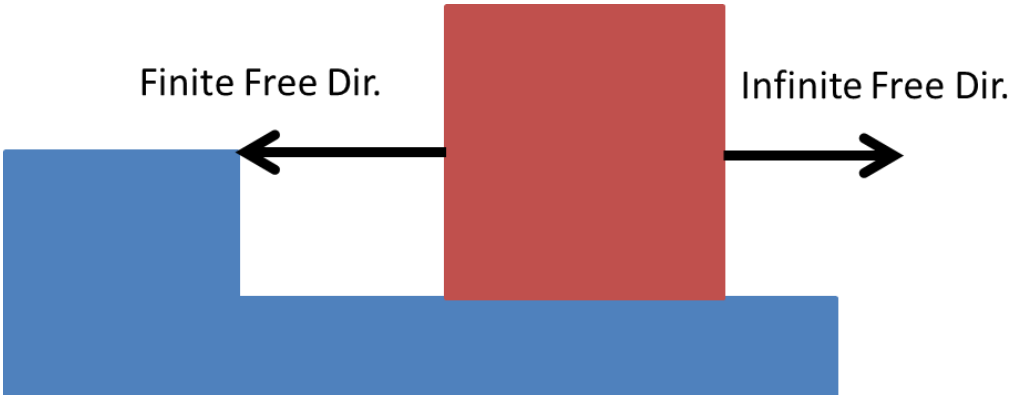


Figure 5: Illustration of finite and infinite free directions

IDENTIFYING FREE DIRECTIONS

There are two types of free directions, finite free directions and infinite free directions. An infinite free direction is a direction a part can move such that it can translate off of the mating part completely without hitting other parts. Thus, the part can translate an infinite distance without hitting any parts. A finite free direction is a free direction that a part can move that will eventually end in the part colliding with another part in the subassembly.

To find the finite free directions the 3D space of all possible translations is discretized with a tessellated unit sphere. While it is impossible to uniformly distribute more than 20 points on a sphere; however, one can get reasonably close. In order to have a nearly even distribution of directions with adjustable resolution, the *icosahedron subdivision* approach is used. [24] This works by dividing each triangular facet of an icosahedron into smaller triangles, and dividing those smaller triangles into even smaller triangles until the desired resolution is reached with each triangle's center point defining a unit vector. This results in a discrete semi-uniform space of all translation space directions. This space is truncated based on the contacts between the two parts. For example, a planar-planar contact removes all free directions not in plane.

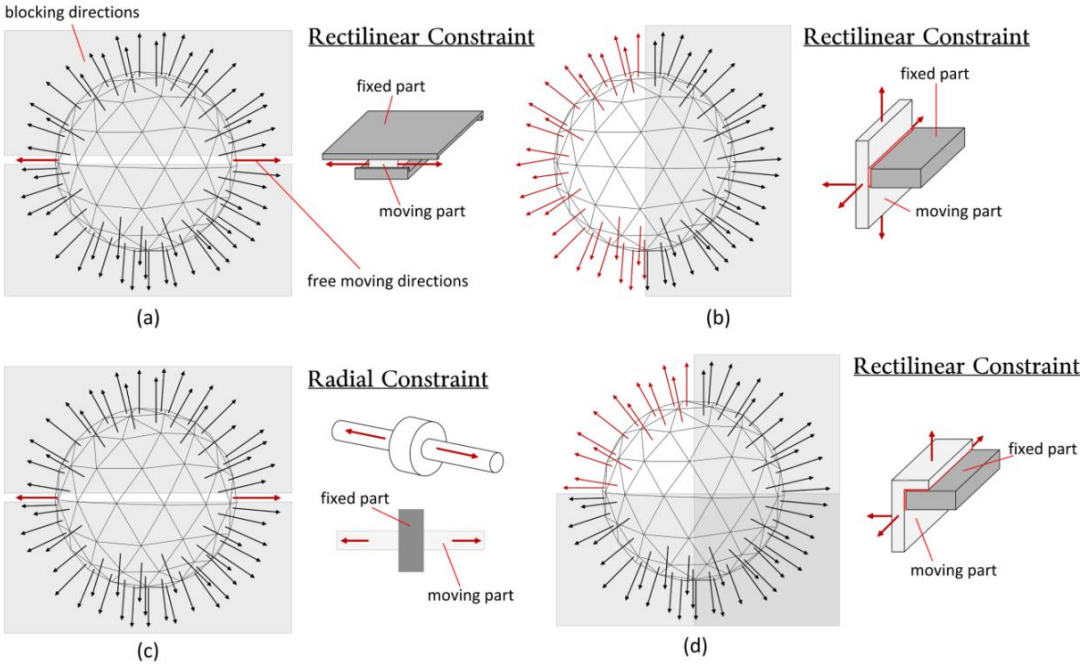


Figure 6: Free directions, based on connection types. **[25]**

Theoretically, a part has the most free-directions in the case of two infinitesimally thin needles in contact resulting in a sphere of free-directions except the direction the

needle is in contact. In reality, this rarely occurs. In the case of a planar contact, the constraint can more or less be described with five directions.

For the case of cylindrical contacts, the normal vectors about the circumference of the cylinder define directions that are blocked. In the case of a simple cylinder in a cylindrical through hole, all but the directions defined by the normals of the cylindrical caps are blocked. Each blocking direction eliminates a half sphere of free-directions as shown in Figure 6.

Using this method proves to be a very accurate way to find free directions between parts. However, it is fairly computationally intensive, making it useful only for small systems where the computation time is short. For the large systems that this research is intended to work with, a faster approach is necessary. To cut down on the computation time, only the six face normals of the oriented bounding box are used in

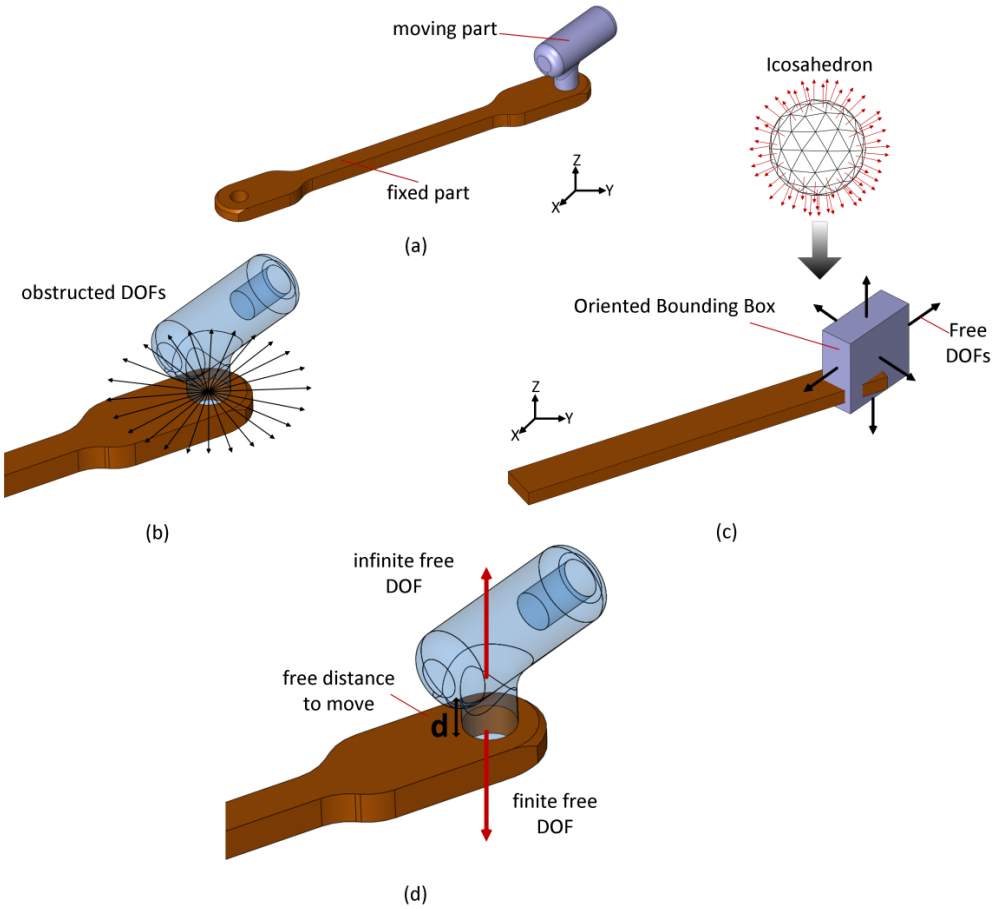


Figure 7: (A)Sample part connection, (B) Obstructed and (C) global free directions (C). finite and Infinite (D) **[25]**

place of the subdivided icosahedron as shown in **Figure 7.**C. While this is not as accurate as using the subdivided icosahedron, a significant speedup is realized.

Assembly Representation

In order to generate possible assembly sequences, it is necessary to have some means to represent assemblies. The representation scheme is perhaps the most important part of the system, as it is required by the geometric reasoning, rules, and evaluation phases. The assembly representation needs to have a means of representing the geometric constraints that exist between parts, the types of connections that exist between parts, and part information. Geometric constraints define the possible motions a part can be carried through in an assembly process. In 3D space with rotations, there is a near infinite amount of possible motions a part can make. It is not possible, nor desirable to represent this entire space of possible part movements.

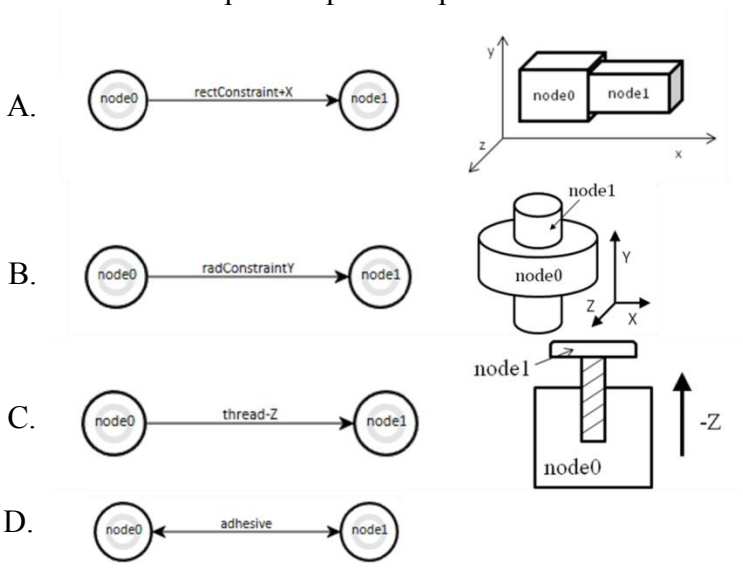


Figure 8: Examples of Constraints from Previous Representation Scheme, figures excepted from [26]

A previous means of representing assemblies developed by Agu [26] for a disassembly planning system was to represent parts as nodes and constraints or connections between these parts as arcs. These arcs contained connection and constraint specification with respect to a global coordinate frame. In order to limit the analysis

requirements, parts were assumed to translate off of the assembly along one of the three coordinate axes. Constraints between parts were specified as labels on arcs that described constraint type and direction. Direction is specified as a 'directional suffix' consisting of a letter denoting which coordinate axis is used and a positive or negative sign denoting the direction along the axis the part is constrained. For example, a rectangular constraint where node0 is prevented from moving in the positive X direction by node1 would be represented by an arc going from node0 to node1 with a "rectConstraint+X" label as can be seen in Figure 8A. For the case of a radial constraint, only one of the three axes(X,Y,Z) is needed, and the arc direction defines which component surrounds the other component. For example, a radial constraint about the Y axis with node0 surrounding node1 would have an arc going from node0 to node1 with the "radConstraintY" label. An example of this connection can be seen in Figure 8B. Threaded and press-fit components are represented much in the same way with the arc going from the component that the other component threads/press-fits into and with a label that specifies which of the six directions it must translate from. An example of this connection can be seen in Figure 8C. This means parts can translate onto to the assembly from an infinite distance from a single direction.

This approach is inherently limited, as it overly simplifies the directions real components are capable of. Another problem is that the way constraints are specified is redundant; the representation required that for every connection to a component, there should be a reverse (same in the case of threaded/press-fit) connection going from a component. In order to improve upon this approach, rather than using global directions, local directions are used. This means instead of using global coordinate axes, we define constraints for a connection with respect to an arbitrarily chosen fixed part. In addition, we add the capability to represent subassemblies. An arc still represents a connection

between two parts; however, the need for a reverse arc is eliminated. The direction on the arc points to the part that is fixed and to which all free directions are referenced. The directions a part is free to move in, or free directions, are primarily used instead of blocked directions (rectangular constraints). However, we maintain the ability to use blocked directions through the use of 'unfree' arcs, or arcs containing blocked directions.

An assembly is represented as a set of nodes, which represent parts, and arcs, which represent connections between parts.(See Figure 9) Three different types of graph elements are used in the representation of an assembly: nodes, arcs, and hyperarcs. Furthermore, collections of parts or subassemblies are represented with hyperarcs (arcs

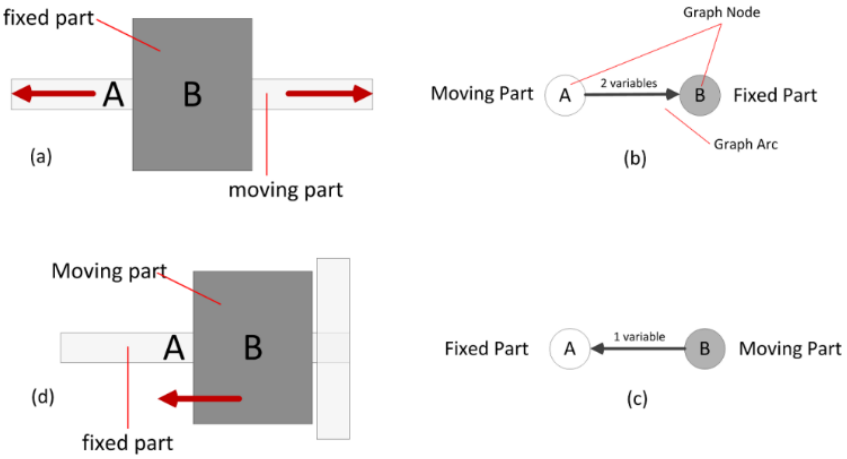


Figure 9: Representation

that connect more than one node together). Additional data can be stored in any of these elements; as local labels and local variables. Local labels are strings used to identify the

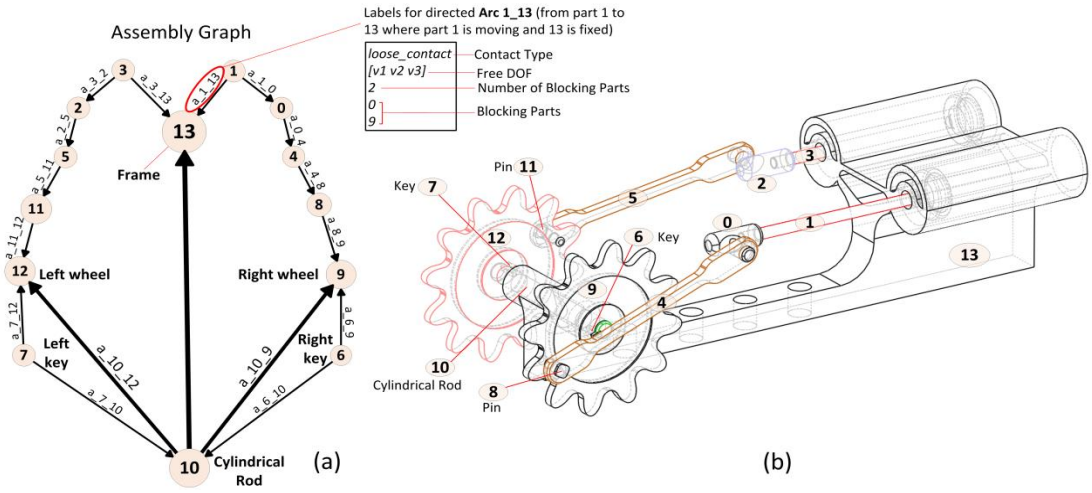


Figure 10: Graph(A) and the assembly it represents(B) **[25]**

connection or part type. The local variables are a list of floating point numbers. This is primarily used to list the free or blocking directions a part is able to move with respect to the other part. These are stored as vectors on the arcs. There are several types of information stored in local variables on the arc, free directions, unfree directions, blocking parts, connection locations, and orientation times. Unfree directions are directions where a part will be blocked by another part. In the free direction vector, in addition to the direction a part can move, parts that block motion in this free direction are also stored. In the preprocessing stage, this information is extracted from each arc and turned into an arc with corresponding unfree directions going to the blocked part for book-keeping purposes. Finally, invisible directions are directions a part can move in with respect to the fixed part, but only for a finite distance inside a part. An example subassembly and graph is shown in Figure 10.

Graph Grammar Rules

In order to find an assembly sequence, we need a means of describing assembly operations. Graph grammars are used to represent feasible operations that can be performed during the assembly process. The graph grammar is used to define transition states in the search tree. A graph grammar rule is an algorithm that is used to find a

Figure 11: A graph rule and its associated applications

graph substructure in a graph and modifies the substructure. Graph grammars consist of three parts, the right hand side, the context, and the left hand side. The left hand side contains the graph substructure to be found, the context contains graph elements present in both the left and right hand sides, and the right hand side specifies how the graph substructure is to be modified.

A very simple graph grammar rule is shown in Figure 11. The rule finds a node (the blue circle) and adds two arcs (arrows) with nodes on them. Repeated applications of the rule on a seed graph consisting of a node, will lead to the formation of a tree-like structure as shown above. The act of finding the graph substructure found in the left hand side is called recognition and the act of modifying said graph substructure is called application. The location in which a rule was recognized and its associated rule application are known as an option. A graph modified by a graph grammar is called a candidate. In order to simplify the presentation of grammar rules only the right and left hand sides of rules will be shown. The GraphSynth software was used to implement graph grammars in this paper. In GraphSynth, collections of rules can be organized into

sets of rules or "rulesets." Additional logical conditions can be applied to these rulesets to carry out actions such as changing rulesets or stopping execution. In GraphSynth, graph grammar rules can contain additional recognition conditions and additional application functions specified by C# code.

In the previous approach [26], disassembly operations were used to deconstruct assemblies. While this work was intended for disassembly planning, it is also applicable to assembly planning; reversing a disassembly sequence results in an assembly sequence. This old approach works by finding parts that are free to move and removes them from the assembly. The problem with using this approach for assembly planning is that it is limited in the diversity of assembly plans it can create. All assembly plans generated by this approach will feature single parts being put on to a single assembly. It cannot generate assembly plans where multiple subassemblies are assembled in parallel and then combined into the final assembly. The approach described here solves these problems. Instead of using disassembly, an assembly approach is used to build up sub-assemblies from smaller sub-assemblies.

PREPROCESSING RULES

It is necessary to preprocess the graph for housekeeping and error removal purposes. A hyperarc is added to every node that is not a fastener. Part blocking information is originally stored in the arc associated with the part that is blocked. This makes it hard to reason about blocking parts; in order to determine if a part is blocking another part, one would need to search every arc on the graph. In order to solve this problem unfree arcs are added. Unfree arcs are exactly like free arcs, except that they carry directions in which parts are blocked by other parts. The geometric reasoning is far from perfect and occasionally encounters connections that cannot be reasoned about. If

these connections were to be processed, they would prevent a feasible assembly sequence from ever being found. One case that occurs is an unknown connection, or a connection that the geometric reasoning is not sure of and has no free directions. As it has no free directions, it will cause all connection feasibility tests to fail. These connections are simply removed from the graph, as it is better to find an assembly sequence that is somewhat valid than it is to find no assembly sequence. Further work is needed to improve geometric reasoning.

RULES FOR FEASIBLE ASSEMBLY OPERATIONS

In order to generate feasible assembly sequences, a set of graph grammar rules was developed to define possible assembly operations. The rules essentially build subassemblies that are further assembled together until all parts are in one subassembly (the finished product). This allows reasoning about assembly operations to be done in parallel with subassemblies.

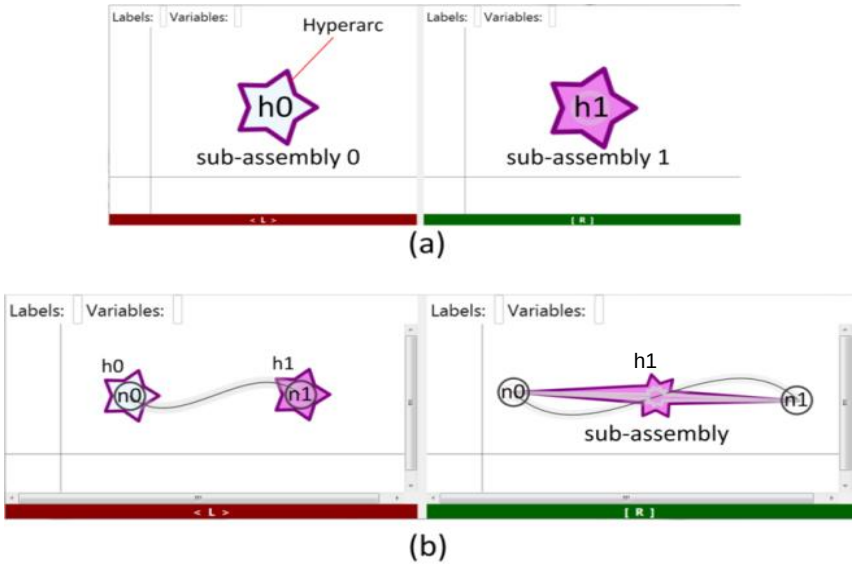


Figure 13: Add Hyperarc(A) and Grow Hyperarc(B) Rules

In the graph, subassemblies are represented as hyperarcs, or arcs that connect to more than two nodes. Reasoning about assembly sequencing is currently done by three graph grammar rules: *add hyperarc*, *grow hyperarc*, and *merge hyperarcs*. The add hyperarc rule, shown in Figure 13, takes a component that is not part of any hyperarc (which has the label "part" on it) and makes it into a subassembly by encoding it with a

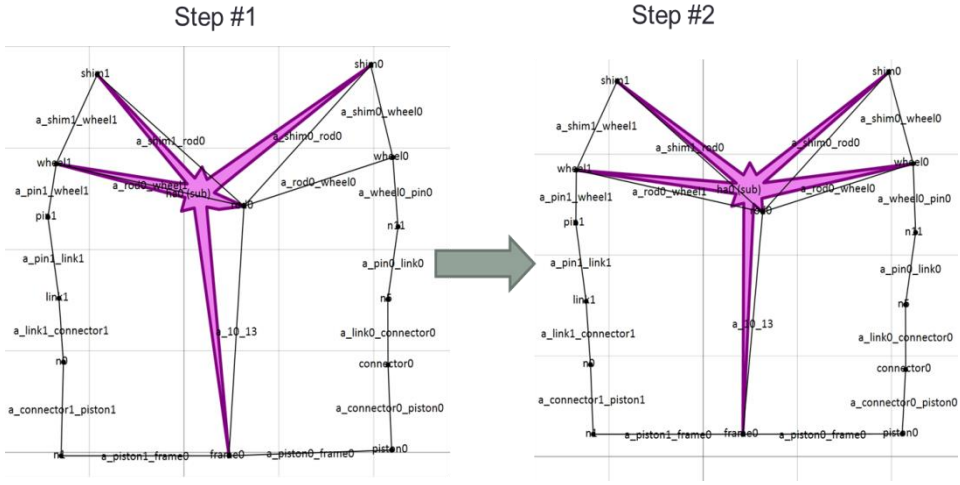


Figure 12: Example of grow rule application

hyperarc (see h1 on the right hand side in Figure 13.a). This is to prevent subassemblies from starting on components such as fasteners. It is important to note that the clear hyperarcs on the left hand side of rules shown above (see h0 in Figure 13.a) indicate that

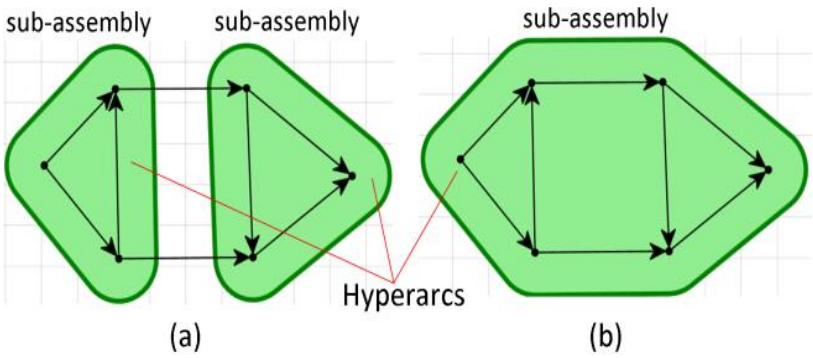


Figure 14: merge hyperarc rule.

said hyperarc must not exist in order for the rule to apply. It is a negative element in terms of grammar recognition. The grow hyperarc rule, shown in Figure 13.b, identifies a node that is not part of any hyperarc, but is connected by an arc to another component that is part of a hyperarc, and brings it into the subassembly. This corresponds to adding a component to a hyperarc. An example of this operation on the graph is shown in Figure 12. The final rule shown in Figure 14 takes two hyperarcs that are connected by an arc that is not an unfree arc and makes them into a single hyperarc. This corresponds to putting two subassemblies together into one subassembly.

As part of the recognition processes for the merge hyperarcs and grow hyperarcs rules, assembly operation feasibility checks are carried out to prevent the rules from recognizing infeasible assembly operations. In order for an assembly operation to be feasible, a component or subassembly must have at least one free direction shared among all arcs going to the subassembly it is being added to. If no free direction exists, then two

subassemblies cannot be joined together into a single subassembly. Thus, the feasibility check fails, preventing rule recognition, if no free direction is found.

Figure 15.a illustrates a scenario in which the rules will not apply because the free directions between the two subassemblies go in different directions.

In addition to assembly operation feasibility checks, an assembly sequence feasibility check is carried out after no rules can apply to ensure that the sequence of assembly operations is valid. In order for a sequence to be feasible, it should result in a graph where all nodes are connected in one final hyperarc. That is to say, all nodes have been merged into one subassembly. Figure 15.b shows an example of one such end-state. When this state is reached, none of these rules will be recognized on the graph, and the process will naturally conclude.

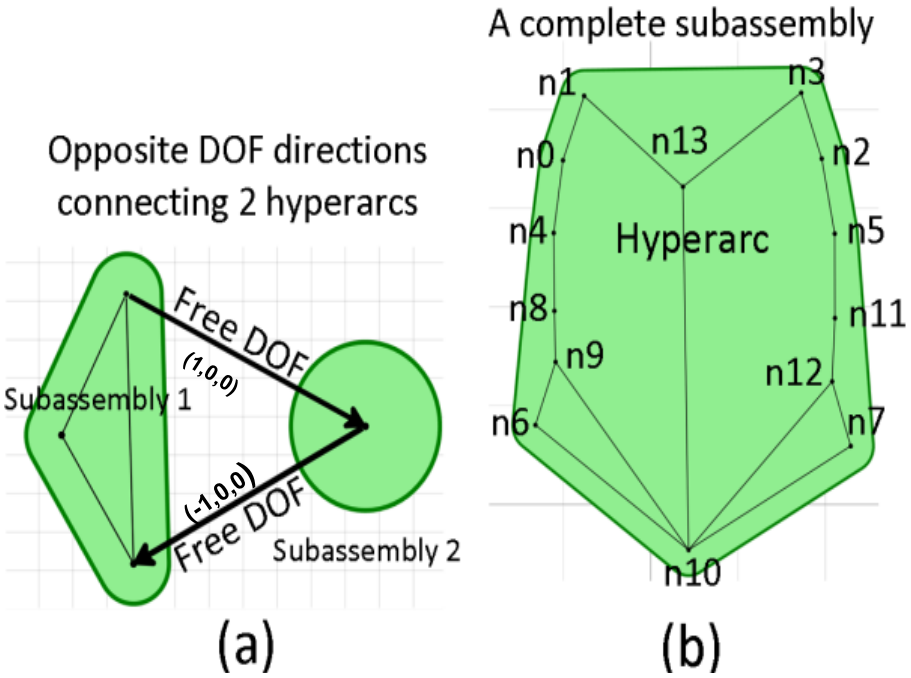


Figure 15: Feasibility Check(A), End Conditions(B)

Evaluation and Search

In order to perform assembly sequencing, one needs metrics to show how the relative cost of one assembly sequence is compared to other assembly sequences. Several metrics are used to evaluate assembly sequences. [25] Time is the most significant metric which is measured. The time metrics is the sum of orientation time, insertion time, and handling time.

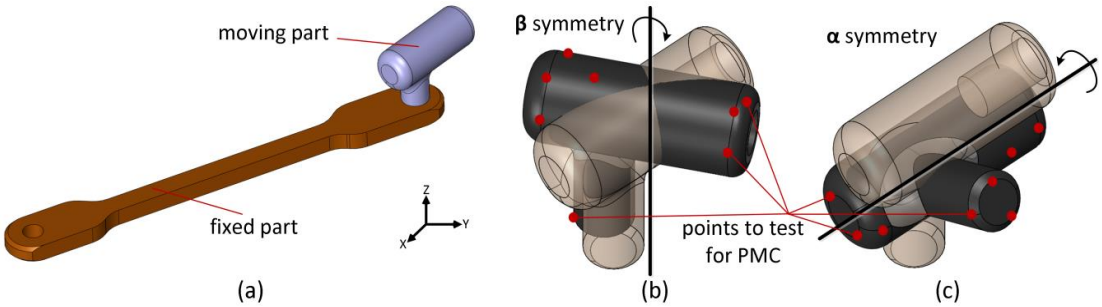


Figure 16: Contacting Parts(A), β symmetry axis(B), α symmetry axis(C) **[27]**

ORIENTATION TIME

Orientation time is the time required to rotate the part into the correct orientation needed for assembly. Boothroyd and Dewhurst [1] have shown that orientation time is directly related to the rotational symmetry properties of a part about two axes. The "β" axis is defined as an axis parallel to the insertion vector. The "α" axis is defined as an axis perpendicular to the insertion vector. The geometric reasoning phase finds the minimum angle through which the part must be rotated in order to look the same about these axes. This procedure is illustrated in Figure 16. By summing up the "α" and "β" angles divided by 720, a relative measure of the orientation time is obtained.

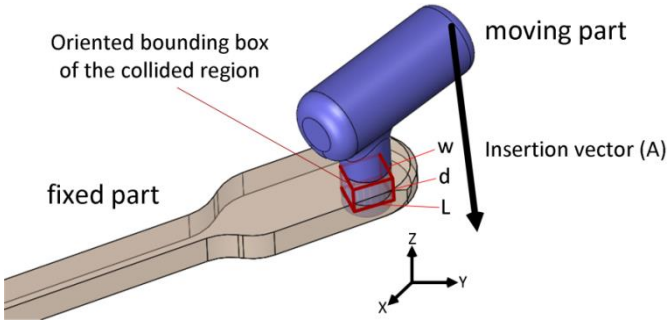


Figure 17: Insertion Time Calculation

INSERTION TIME

Insertion time is the time required to insert a component into another component. To calculate insertion time, the geometric kernel is used to find the bounding box of the collided region. The length of the bounding box edge parallel to the insertion direction is used to determine the length of overlap or insertion.(See Figure 17) By multiplying this by a velocity (e.g. cm/s) a rough estimate of insertion time can be obtained.

HANDLING TIME

Handling time is the time it takes to handle a part. Handling time is determined by the size, shape, and weight of the part. Very small, hard to grasp, or heavy parts are difficult to handle. In practice, it is very difficult to give an objective measure of grasping difficulty of a part, so only size and weight will be used. Size is calculated from the largest dimension of the oriented bounding box. Weight is taken simply as the volume of the part multiplied by an arbitrary density. Given a specific part material, weight could be more accurately calculated.

$$\text{Handling Time} = 0.0125 * Weight + 0.011 * Weight * Size$$

SEARCH

In order to find an assembly plan that is feasible and optimal, the space of all possible assembly sequences must be searched. The grammar rules define operations that can be performed at a given step in the assembly sequence. The grammar rules are run and applied everywhere they are recognized, to generate candidates for each of the possible assembly operations. These candidates form the children nodes of the assembly tree.

One problem that arises is that the search can produce assemblies that prevent some of the components from becoming part of the final assembly. For example, a box is built around the location where another subassembly must be assembled, before said sub assembly is in place. To prevent this, the recognition part of a rule is used unconventionally; it is used to modify the graph.

Traditionally, a graph is checked for locations where a rule is valid in a recognition phase and then modified in an application phase. In the recognition phase of the merge and grow rules, the graph is also checked for the existence of impossible assembly operations as specified in the rules section. If these conditions are found, then there is no way all parts can be put together into one subassembly and the search down the corresponding branch of the tree should not be pursued further.

If such conditions are detected, the additional recognize functions assign a global label to the graph that prevents recognition of all other rules, causing the search to terminate. This saves time over the conventional approach of recognize and apply, as the search would go through a potentially lengthy recognition phase before performing the same calculations performed in the recognition phase.

To find the best assembly plan, ordered depth first search was used. Ordered depth first search is a tree search algorithm based on depth first search. Unlike traditional

depth first search it orders children nodes based on their scores. Among the scoring metrics used were total time and condensed time. Total time is the amount of time carried out among all assembly operations. Essentially, this is how many person-hours are required to assemble a product. Condensed time is the overall time that assembly takes from start to finish. One might minimize condensed time to build the product faster at the expense of increasing total time by carrying more operations in parallel.

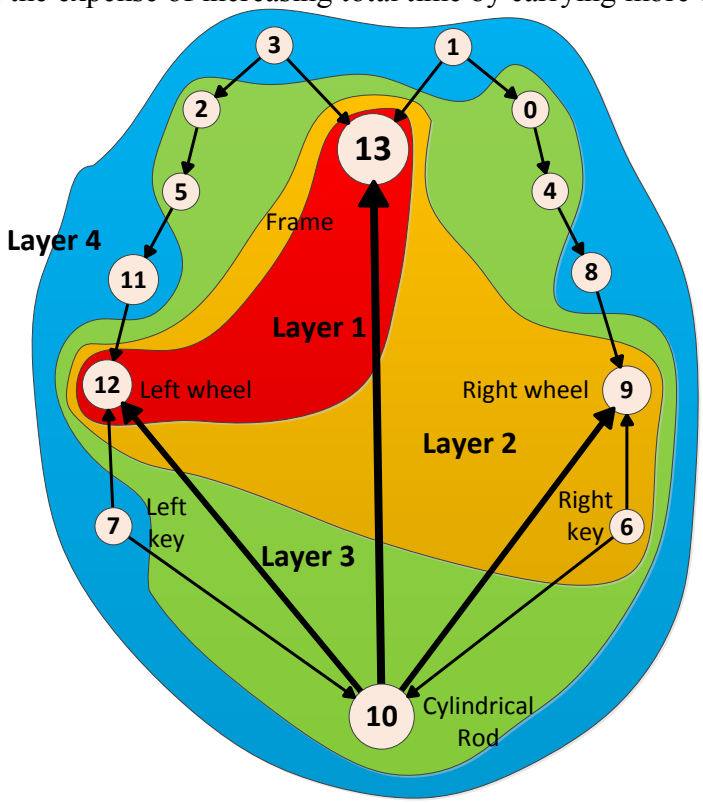


Figure 18: Layering

LAYERING HEURISTIC

In practice, the search process is relatively slow due to large numbers of invalid assemblies. To reduce the runtime, it is necessary for the search to have better information about what lies ahead. In order to better inform the search process, a heuristic is used. Parts in an assembly exist in layers. Parts on the outer layers must go

on last, but are essentially free to move off of the assembly. Parts present on the layer beneath the top layer are also free to translate off of the assembly if the outer layer is removed. We know that parts on the lower layers should probably come on to the assembly sooner than the parts on the outer layers, making this a useful heuristic. Figure 18 illustrates layering information for an example assembly.

Before we perform search on an assembly, we make a copy of the graph, disassemble it with graph grammar rules, and then tag the parts on the original graph with layering information. Three rules in three rulesets are used to accomplish this. The first ruleset checks to see which parts are free to be removed, the next ruleset removes them, and the last deletes tags on the arcs, so that the process can be repeated. The first rule finds a part in the graph that does not have the label 'checked' and checks if the part is free to be removed. To do this, a boolean intersection of the free direction vectors present on all arcs is performed. All unfree directions are then boolean subtracted from this result and if there is at least one vector present, then the part is free to move. The labels "free" added to the part's local labels if this is the case. Regardless, of whether the part is free or not, the "checked" label is added to the part's local labels. Because the rule cannot be recognized on parts with the label "checked," all parts will be checked. Once no more rules are found to recognize, the program advances to the next ruleset. The next rule recognizes only parts that have the "free" label, and removes all arcs connecting to the part, effectively removing parts from the subassembly. The program adds all options to a layer list and then applies all options in a single step. The next ruleset simply removes the "checked" label from all parts and returns to the first ruleset. After the part has been completely disassembled, the remaining parts (if any) are added to the layer list as the lowest layer. The program then adds layer tags to all parts in the original graph, which is used by the search as a heuristic. Parts with a lower layering count, or parts on

the bottom layer, are preferred over parts which have a higher layering count. This is because parts on the bottom layers, which were removed last by the disassembly rule, should probably be assembled before parts on the upper layers to prevent the search from building assemblies that prevent the assembly of all parts.

Results

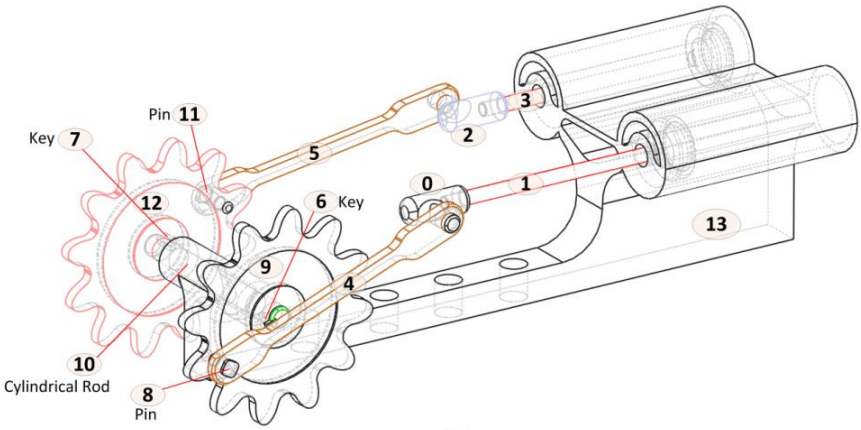
A batch script is used to control different parts of the program. The batch script first deletes previous run information and places the input CAD file in the correct location. The batch script then starts the geometric reasoning process, which outputs a graph describing the CAD model and point clouds for convex hull determination by the evaluation system. The batch script moves these files in to the input directory for the search system. The batch script starts up the search program. A preprocessing ruleset is run first so that the graph is in the proper format for the following steps. Next the graph is tagged with layering information. Then the part is run. This process was found to be very dependent upon the fidelity of the CAD models of the individual parts used to build the whole.

An example of an assembly plan generated with this approach and its associated assembly is shown below. The part shown below was derived from the GrabCad online CAD repository. The part was converted from its native SolidWorks format to the Parasolid format required by the geometric reasoning engine. In Figure 19 one can see the optimized assembly plan, the operations that must be carried out, and the estimated time it takes to carry out these operations.

The first two operations consist of putting the pistons 1 and 3 onto the frame 13 and are erroneously marked as taking zero seconds to complete. This is due the fact that in the CAD model the pistons are not contacting the frame, preventing the calculation of assembly time. Overall plan time is estimated to take around 47 seconds, which is realistic given that the CAD model is small. Total time to load in the CAD model and produce an optimized assembly plan was 91 milliseconds.

The assembly sequence operations carried for this plan are shown in Figure 22 in the appendix. The plan generated was found to be physically feasible, indicating that this approach generates valid assembly plans. Other parts were attempted with this approach, and were found to never complete. This could be due in part to the presence of improper constraints induced by the presence of press-fit components.

One such example of an assembly for which no plan could be found, shown in



(a)

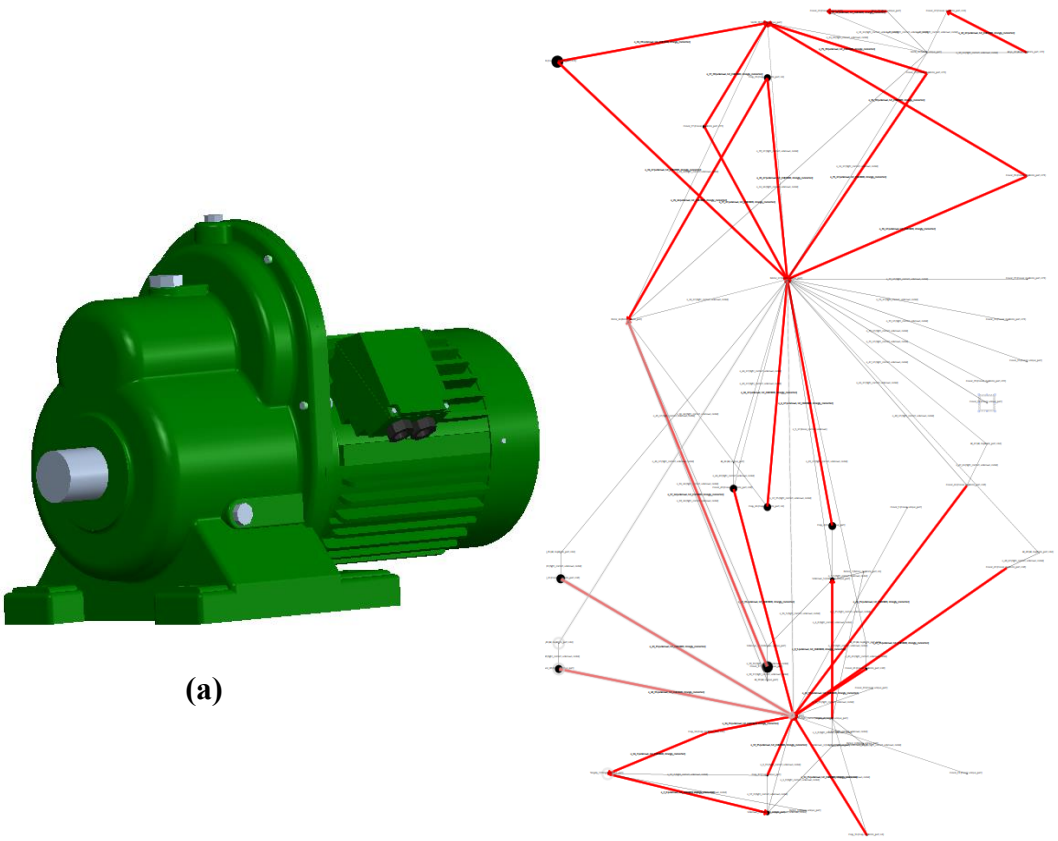
```
========= Plugins ==========
1. Beam Search
2. Best First Search
3. Ordered Depth Search
4. Hill Climbing
5. Depth First Search
Push the key of the plugin you would like to run (F1 for other commands).
 >3

Optimized Assembly Plan:

 Step 1: put part n3 onto part n13      time = 0
 Step 2: put part n1 onto part n13      time = 0
 Step 3: put part n4 onto part n0       time = 2.20000149527317
 Step 4: put part n11 onto part n12     time = 3.2
 Step 5: put part n8 onto part n9       time = 3.2
 Step 6: put part n10 onto part n13     time = 4.09712686292756
 Step 7: put part n10 onto part n12     time = 4.64712686292756
 Step 8: put part n7 onto part n12      time = 4.67150806864998
 Step 9: put part n6 onto part n10      time = 4.67150806864998
 Step 10: put part n2 onto part n3      time = 5.67722691163777
 Step 11: put part n11 onto part n5     time = 4.67751318231775
 Step 12: put part n0 onto part n1      time = 5.67722691163777
 Step 13: put part n8 onto part n4      time = 4.67122446702541

Total evaluation time = 47.390462831047 sec
Processing time = 91 millisec
```

(b)

Figure 19: Example part(A) and its assembly plan(B) **[27]**

Figure 20.a, is a motor with a gearbox downloaded from an online CAD repository [27]. Several problems were found that may have prevented an assembly sequence from being

found for this part. First, there is a gear inside the gearbox that is completely enclosed by the gearbox case, making it impossible to assemble with traditional manufacturing. This case is illustrated in Figure 21.a. It is likely that the only way to manufacture this assembly is for part of the gearbox case to be welded together after placing the gear inside. The designer may have designed the assembly this way because they reverse engineered the assembly and may have failed to capture the fact that the part must be

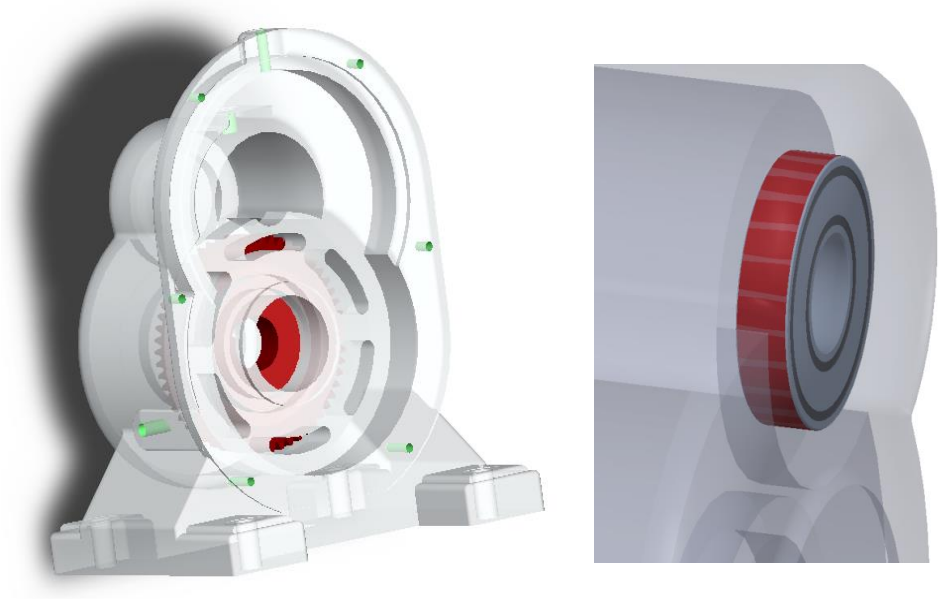


(a) **(b)**

Figure 20: Example assembly (A) for which no plan could be found and its associated graph with incorrect connections highlighted in red

welded due to the destructive nature of the reverse engineering process. Second, the graph generated from the CAD model of this part was found to contain arcs that do not

contain any free directions. These connections are highlighted in red in Figure 20.b. All connections lacking free directions were found to occur between parts that with intersecting geometry. This makes finding free directions impossible as contact faces cannot be determined when parts intersect each other. One case where intersecting geometry occurs is with press-fit components, as designers most often represent press-fit components in their un-deformed configuration. All of the bearings, being press-fit components, had this problem. As shown in Figure 21.b, the bearing's outer diameter(highlighted in red) is larger than the hole its placed in as indicated by the white stripes on the red bearing.



(a) **(b)**

Figure 21: Example of component(highlighted in red) that cannot be assembled(A), example of parts with intersecting geometry(B)

PROBLEMS WITH THIS APPROACH

This approach has several problems. It is not possible to find out if an assembly sequence will ever be found. If no possible assembly sequence exists for the given assembly, the search process will loop forever, attempting to find a solution. The search also experiences an increase in runtime when large amounts of invalid solutions are generated.

SHISKEBAB RULES

When humans perform assembly planning, they often use 'common sense' to come up with assembly plans. One common feature of many mechanical assemblies is a number of parts stacked on to a cylindrical component. We call this feature a "shiskebab" for its similarity to the food item of the same name. Parts along a cylindrical component constrain each other linearly. Therefore, there are a very limited number of feasible assembly operations to go through. Take the example of a nut and a bolt that holds a stack of plates together. The only feasible assembly operation sequence is to stack each plate, one after the other, and then to put the bolt on through the stacks. It also makes more 'sense' to perform all these steps in order. In the graph preprocessing phase we identify these shiskebabs and treat them much like subassemblies in the graph reasoning phase. If a subassembly grows or merges to include a component that is part of a shiskebab, then we should merge on the rest of the shiskebab and then later interpret this operation in the evaluation phase.

DISASSEMBLY APPROACH

The forward approach of assembling a final subassembly from many different subassemblies was found to be computationally inefficient. Large amounts of time are spent building up assemblies that prevent other parts from being added, necessitating the search process to work around 'dead-ends'. One way around this is to use a backwards approach, by disassembling the CAD model, one can avoid all of these invalid results.

A disassembly approach based on the Non-directional blocking graph method is described here. First, all direction vectors present in the graph are extracted and placed on individual arcs going from a single node that does not represent a part. We call this structure a compass rose, as it lists all available directions. Next, one of these directions is chosen and set as a global variable. The graph is modified with labels and arc additions such that it represents a Directional Blocking Graph(DBG).

Strongly connected components of a graph are the sets of nodes that have a path that connects them to every other node in the component. By finding the strongly connected components, we can determine which subassemblies can be removed in a given direction. The strongly-connected-component finding algorithm used is Tarjan's algorithm [27] which runs in linear time with the number of edges.

After we find strongly connected components, we merge strongly connected components until we end up with only two hyperarcs or separable subassemblies. This is because typical assembly operations do not use more than two moving subassemblies. The process then repeats on each subassembly until all subassemblies are disassembled into individual parts. Rules for this approach have been implemented and are described in the appendix; however, they have not been integrated with an evaluation system.

Future Work

There is much that could be done with this system. Currently, it is difficult to verify if a given free direction or assembly sequence is valid. One may end up with assembly sequences that are valid to the planner, but are not possible in the real world due to invalid constraint specifications. A part may lack connections or have too many free directions leading the planner to find assembly sequences that require part translation through solid objects. CAD file translation also presented many issues. Some parts were found to be split into separate entities after translation leading to improper constraint specification.

CAD files for parts with components that undergo elastic deformation, like springs and press-fits often resulted in invalid constraints being found. This is due to the fact that many of these components are represented in the CAD file in their un-deformed configurations

Conclusion

An automated assembly sequence planning system using graph grammars was successfully demonstrated. This approach uses a novel graph grammar to capture assembly constraints. A unique assembly representation approach was developed using arcs to store connection information with respect to a reference or moving part. In addition, the approach developed here differs from previous graph grammar based assembly planners in that it is capable of planning with subassemblies. Previous planners using graph grammars could only place a single part on at a time. Furthermore, a new graph grammar based method for assembly sequencing based on disassembly was developed in this thesis. This tool could be used to enhance the DFMA design process by providing the designer with feedback about the assembly process. This tool can provide the designer with an estimate of assembly time for the whole assembly and each assembly operation, allowing the designer to easily optimize a design for assembly. Additionally, this tool also generates assembly plans that can be used to manufacture a product. Together, these features make this tool capable of improving how products are designed and manufactured.

APPENDIX A: EXAMPLE ASSEMBLY SEQUENCE

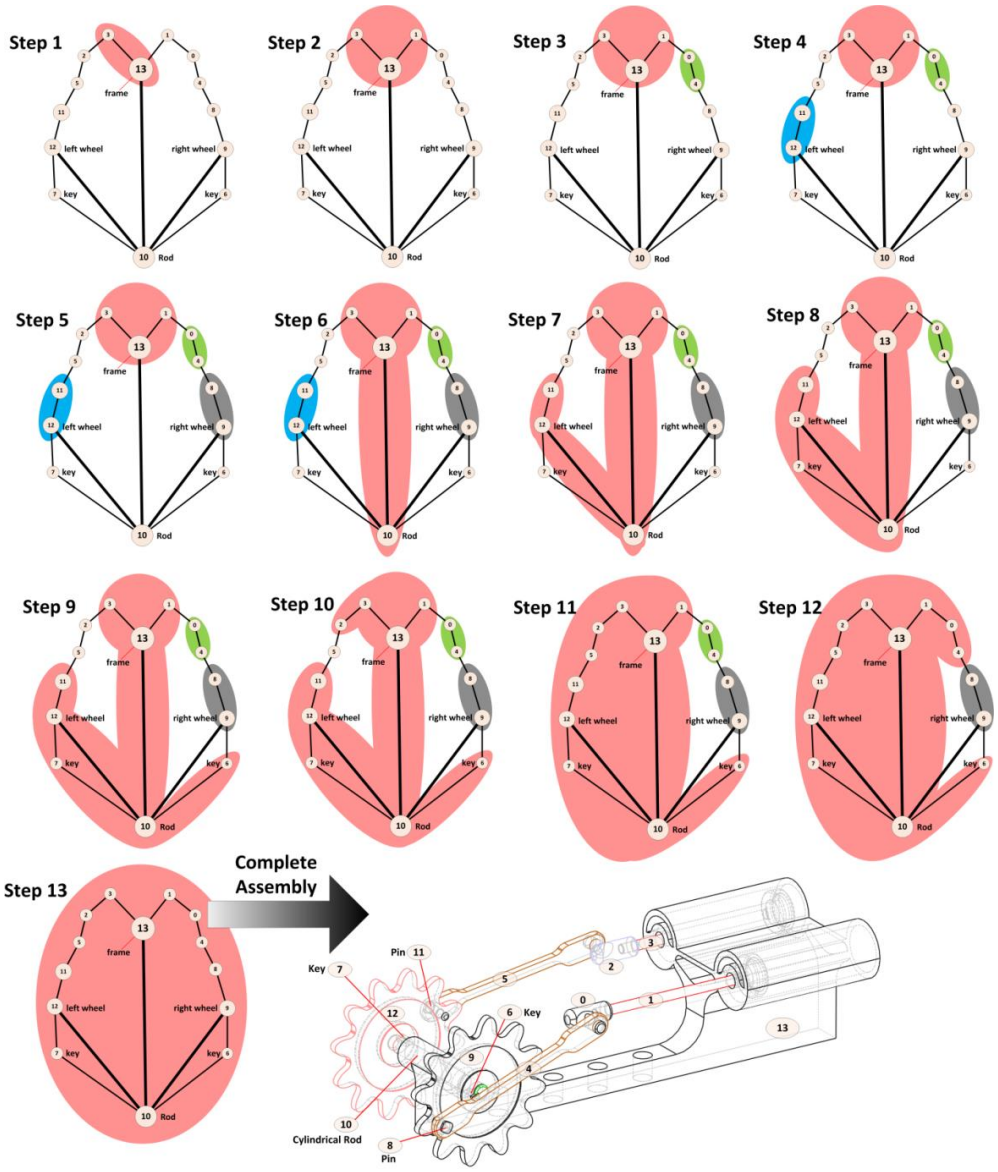


Figure 22: example assembly sequence

APPENDIX B: RULES

Assembly Rules

PREPROCESSOR RULESET

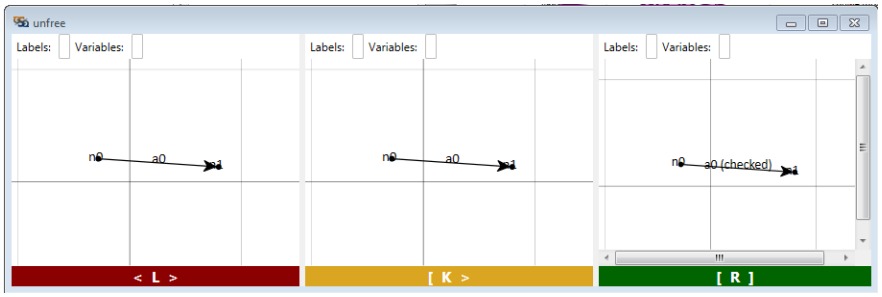


Figure 23: Make unfree arcs

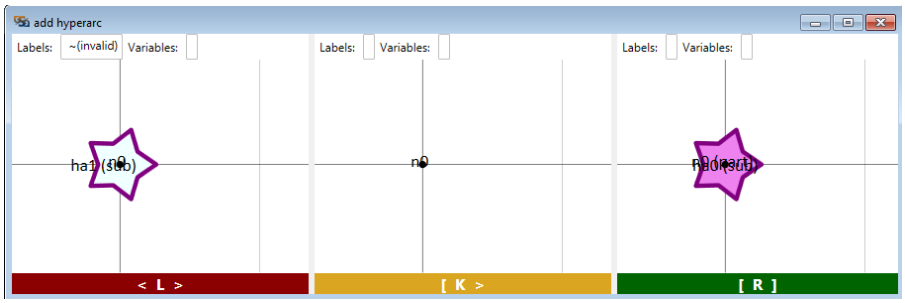


Figure 24: Add hyperarc

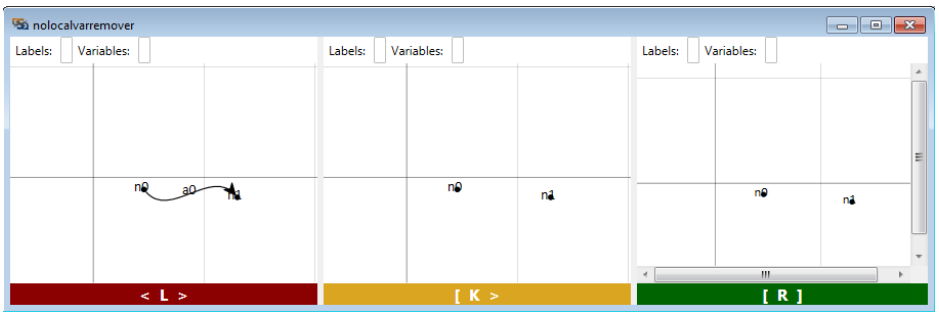


Figure 25: no local variables

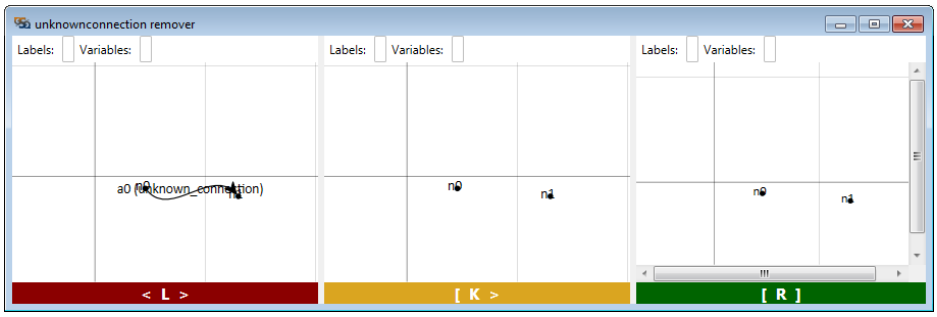


Figure 26: Unknown connection remover

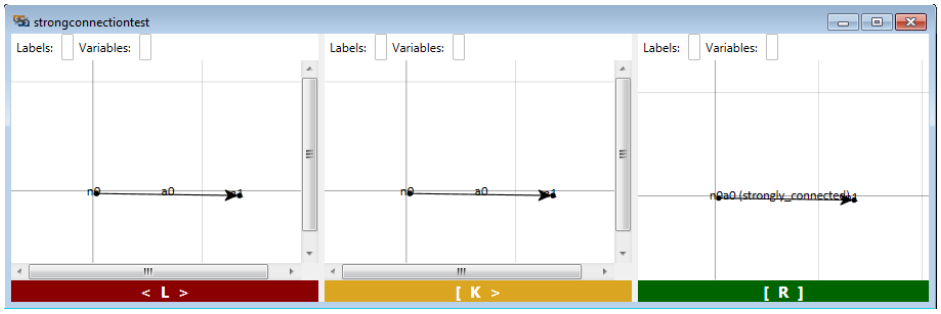


Figure 27: Strong connection test

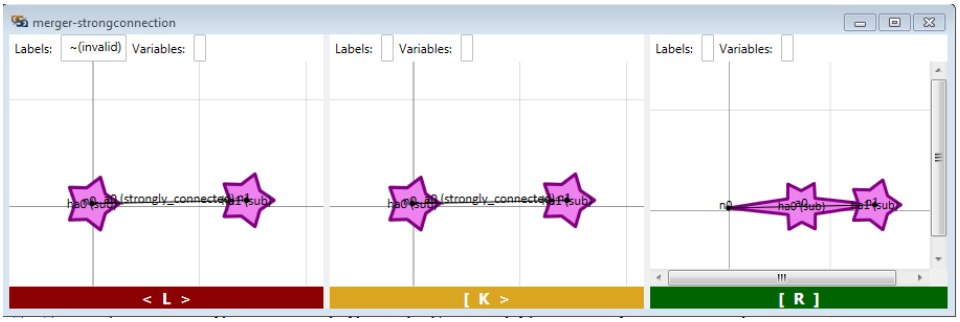


Figure 28: Merger strong connection

Each one these rules is in its own ruleset.

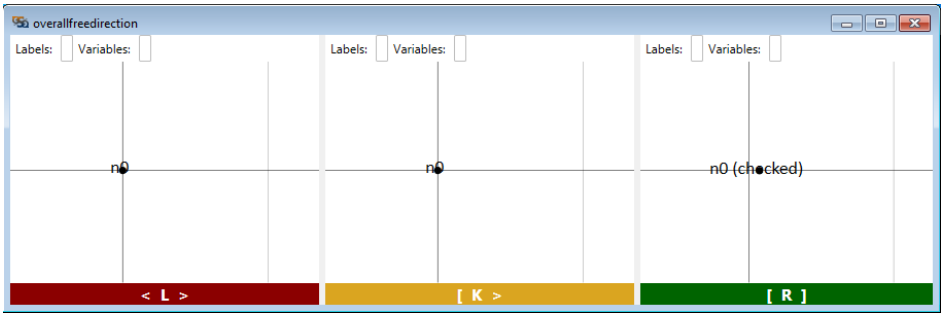


Figure 29: Find overall free direction

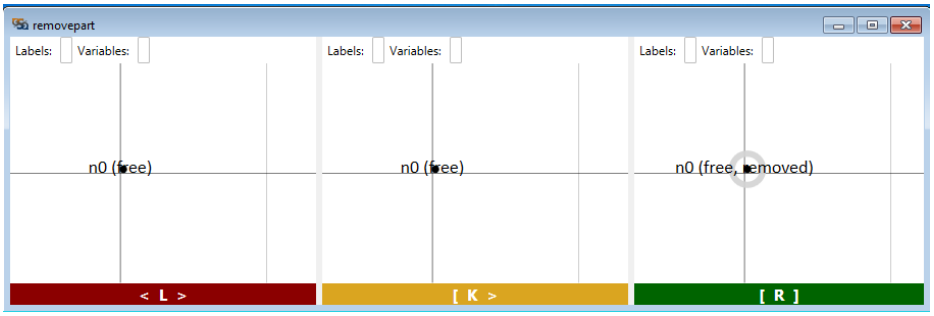


Figure 30: Remove part

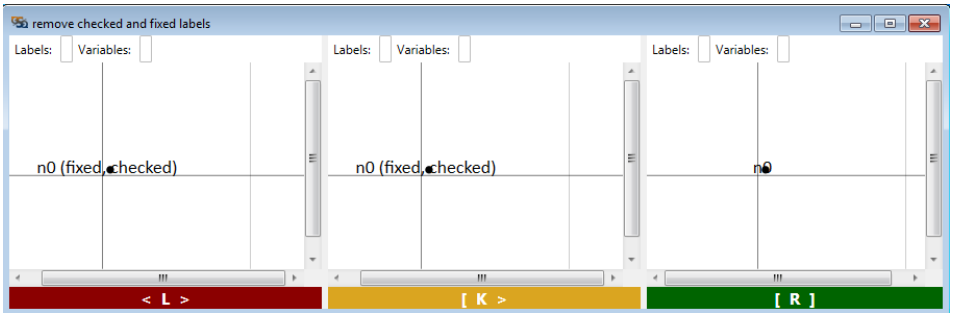


Figure 31: Remove checked

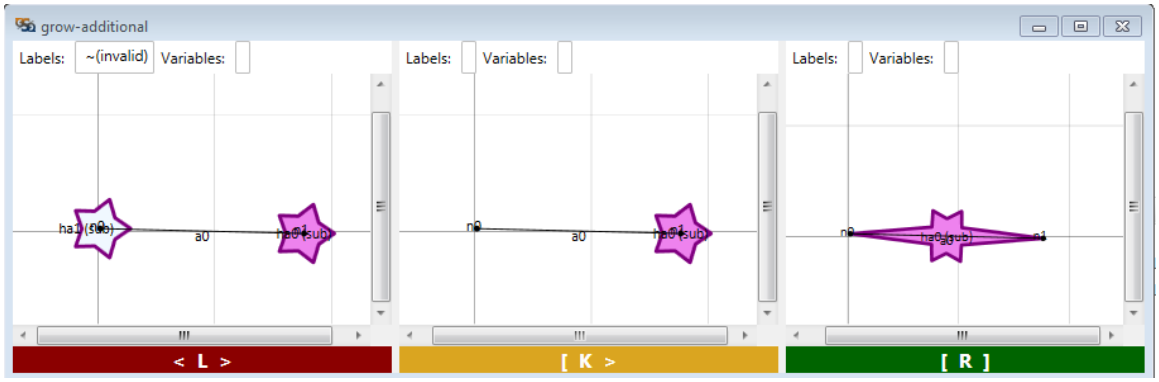


Figure 32: Grow rule

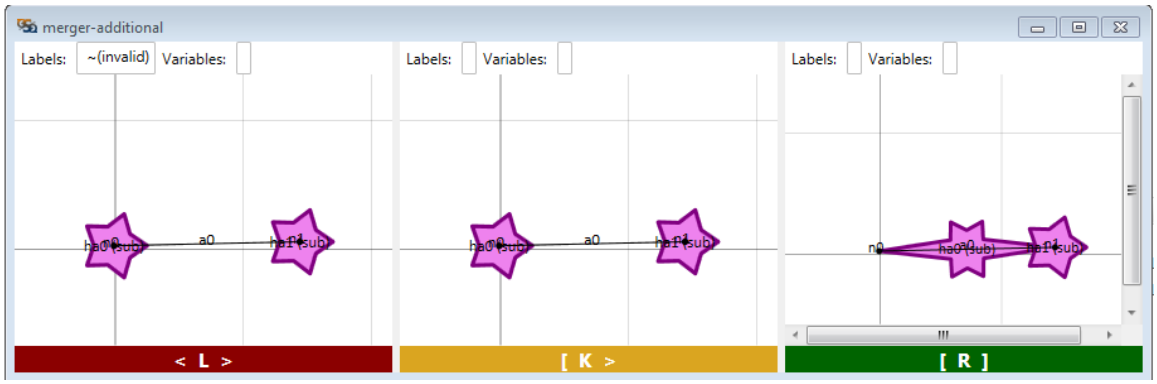


Figure 33: Merger rule

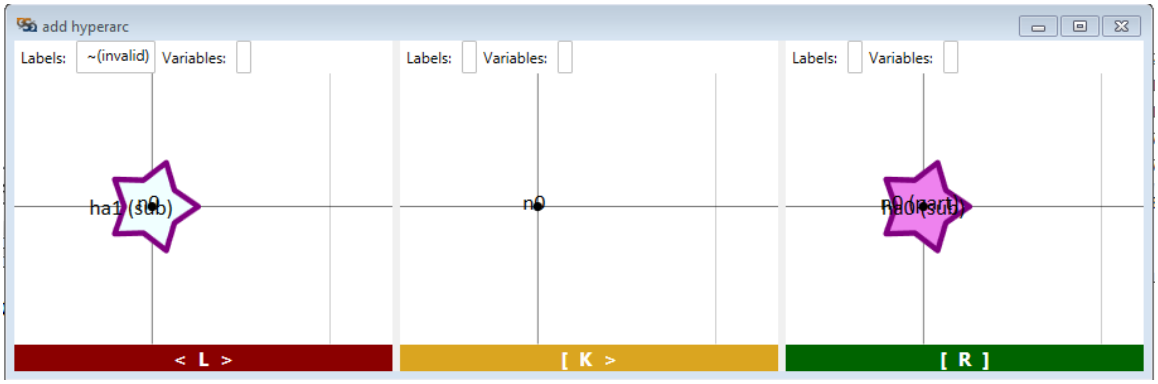


Figure 34: Add hyperarc

Disassembly Rules

DISASSEMBLY RULESET 0: PREPROCESSOR

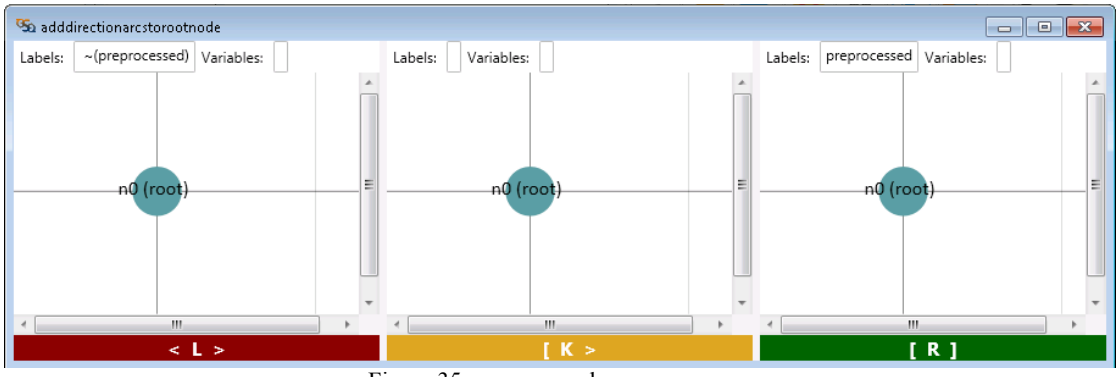


Figure 35: make compass rose

Use additional functions to add directions to the root node to build up a compass rose

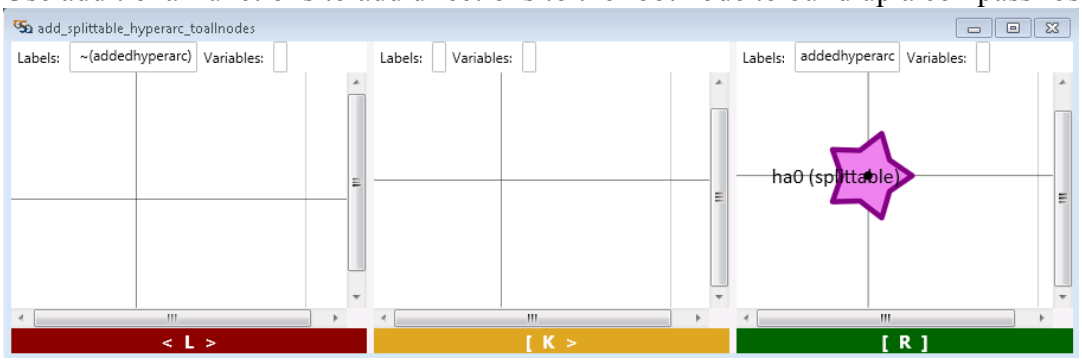


Figure 36: add hyperarc to all nodes

Put a hyperarc that connects all nodes

DISASSEMBLY RULESET 1: DIRECTION SELECTION

Figure 37: choose direction

Choose a direction from the compass rose and use additional functions to add that direction to global variables

DISASSEMBLY RULESET 2: DIRECTIONAL BLOCKING GRAPH (DBG) FORMATION

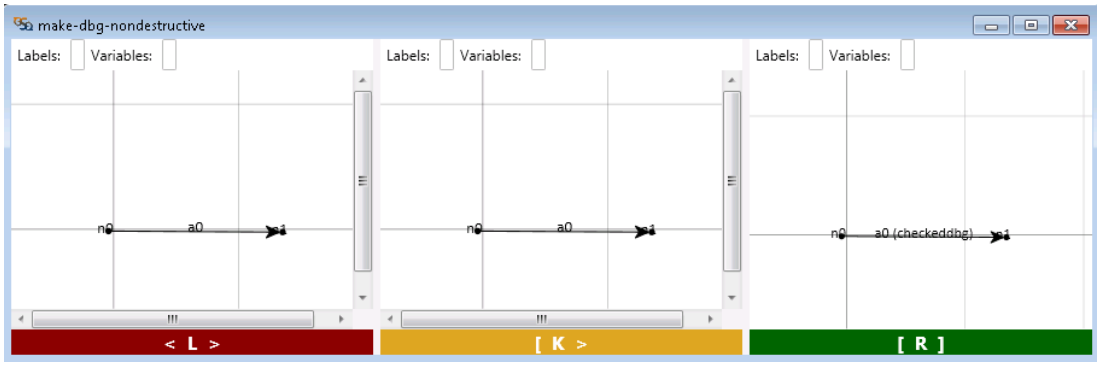


Figure 38: make DBG

For each arc in the graph, add a label or arc, such that the graph now represents a DBG in the direction defined in global variables.

RULESET 3: STRONGLY CONNECTED COMPONENT ANALYSIS

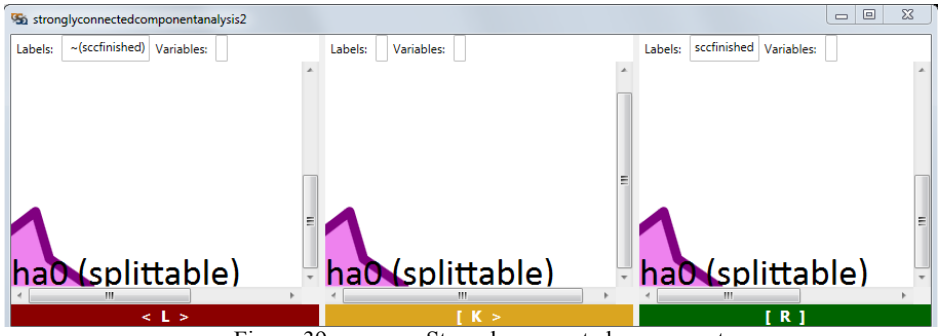


Figure 39: Strongly connected components

Run strongly connected component analysis on a given hyperarc

DISASSEMBLY RULESET 4: PARTITIONING

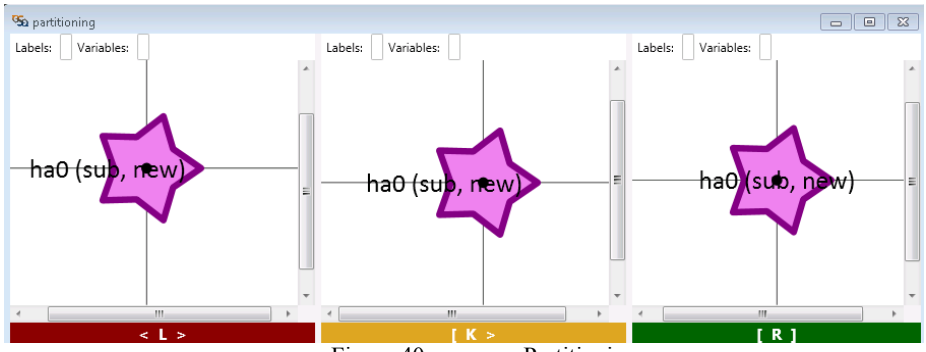


Figure 40: Partitioning

Choose a hyperarc in the graph without any outgoing arcs and merge everything else into one hyperarc.

DISASSEMBLY RULESET 5: RESET

Figure 41: Remove checkeddbg

Remove 'checkeddbg' label so that the DBG can be found again

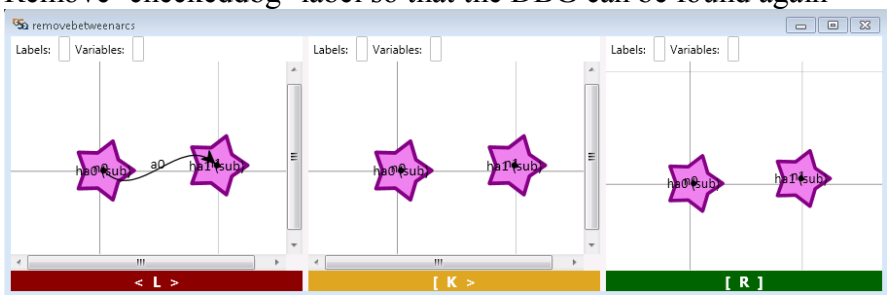


Figure 42: Remove subassembly

Remove any arcs that go between hyperarcs; represents separating subassemblies

Figure 43: Remove fake arcs

Remove any arcs that were added so that the DBG could be represented

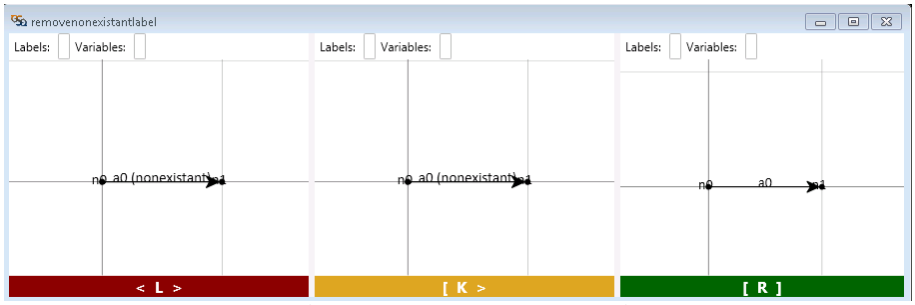


Figure 44: remove nonexistant

Remove the ‘nonexistent’ label that tells strongly connected component analysis to treat an arc like it isn’t there.

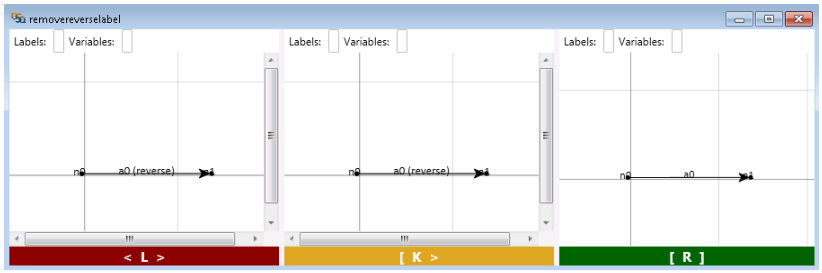


Figure 45: Remove Reverse Labels

Remove the ‘reverse’ label that tells strongly connected component analysis to treat an arc like it is reverse of its current direction.

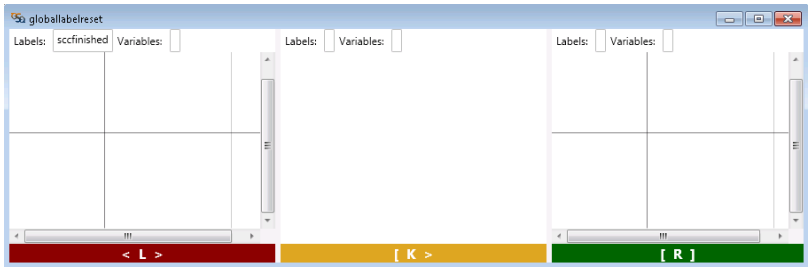


Figure 46: Reset Global Labels

Reset all global labels

BIBLIOGRAPHY

[1] G. Boothroyd, P. Dewhurst and W. Knight, Product Design for Manufacture and Assembly, New York: Marcel Dekker Inc, 1994.

[2] S. G. Kaufman, R. H. Wilson, R. E. Jones, T. L. Calton and A. L. Ames, "LDRD Final Report: Automated Planning and Programming of Assembly of Fully 3d Mechanisms," Sandia National Laboratories, Albuquerque, 1996.

[3] P. Jiménez, "Survey on assembly sequencing: a combinatorial and geometrical perspective," *Journal of Intelligent Manufacturing,* vol. 24, no. 2, pp. 235-250, April 2013.

[4] T. L. De Fazio and D. E. Whitney, "Simplified Generation of All Mechanical Assembly Sequences," *IEEE Journal of Robotics and Automation,* Vols. RA-3, no. 6, pp. 640-658, 1987.

[5] L. Homem de Mello and A. Sanderson, "AND/OR graph representation of assembly plan," *Robotics and Automation, IEEE Transactions on,* vol. 6, no. 2, pp. 188,199, Apr 1990.

[6] L. S. Homem de Mello and A. C. Sanderson, "Representations of mechanical assembly sequences," *IEEE Transactions on Robotics and Automation,* vol. 20, no. 5, pp. 211-227, 1991.

[7] L. Homem de Mello and A. Sanderson, "A correct and complete algorithm for the generation of mechanical assembly sequences," *Robotics and Automation, IEEE Transactions on,* vol. 7, no. 2, pp. 228-240, 1991.

[8] R. H. Wilson and J.-C. Latombe, "Geometric reasoning about mechanical assembly," *Artificial Intelligence,* vol. 71, pp. 371-396, 1994.

[9] D. T. Le, J. Cortés and T. Siméon, "A path planning approach to (dis)assembly sequencing," in *Automation Science and Engineering, 2009. CASE 2009. IEEE International Conference on*, Bangalore, 2009.

[10] S. M. Lavalle and J. J. Kuffner Jr., "Rapidly Exploring Random Trees: Progress and Prospects," in *Algorithmic and Computational Robotics: New Directions*, 2000.

[11] S. H. Huang, "Artificial neural networks in manufacturing: concepts, applications, and perspectives," *IEEE Transactions on Components, Packaging, and Manufacturing Technology: Part A,* vol. 17, no. 2, pp. 212-228, 1994.

[12] R. Rajagopalan and P. Rajagopalan, "Applications of neural network in manufacturing," *Annual Hawaii International Conference on Systems Science,* pp. 447-453, 1996.

[13] C. Sinanoglu and H. R. Börklü, "An assembly sequence-planning system for mechanical parts using neural network," *Assembly Automation,* vol. 25, no. 1, pp. 38-52, 2005.

[14] D. Ben-Arieh, K. R. Ranjan and M. K. Tiwari, "Analysis of assembly operations'

difficulty using enhance expert high-level colored fuzzy Petri net model," *Robotics and Computer-Integrated Manufacturing,* vol. 20, no. 5, pp. 385-403, 2004.

[15] G. Dini, F. Faillil, F. Lazzerini and F. Marcelloniz, "Generation of optimized assembly sequences using genetic algorithms," *Annals of the CIRP,* vol. 48, no. 2, pp. 17-20, 199.

[16] R. M. Marian, L. H. S. Luong and K. Abhary, "Assembly sequence planning and optimisation using genetic algorithms Part I. Automatic generation of feasible assembly sequences," vol. 6, no. 1568, pp. 223-253, 2003.

[17] G. C. Smith and S. S.-F. Smith, "An enhanced genetric algorithm for automated assembly planning," *Robotics and Computer-Integrated Manufacturing,* vol. 18, no. 5-6, pp. 355-364, 2002.

[18] S. S.-F. Smith, "Using multiple genetic operators to reduce premature convergence in genetic assembly planning," *Computers in Industry,* vol. 54, no. 1, pp. 35-49, 2004.

[19] J. F. Wang, J. H. Liu and F. Z. Y., "A novel ant colony algorithm for assembly sequence planning," *International Journal of Advanced Manufacturing Technology,* vol. 25, pp. 1137-1143, 2005.

[20] Siemens PLM Software, "Parasolid," 2012.

[21] J. Erickson, L. J. Guibast, J. Stolfi and L. Zhang, "Separation-Sensitive Collision Detection for Convex Objects," in *ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[22] A. Raabe, S. Hochgurtel, J. Anlauf and G. Zachmann, "Space-Efficient FPGA-Accelerated Collision Detection for Virtual Prototyping," *Design Automation and Test(DATE),* pp. 206-211, 2006.

[23] S. Redon, Y. J. Kim, M. C. Lin and D. Manocha, "Fast Continous Collision Detection for Articulated Models," in *ACM Symposium on Solid Modelling and Applications*, 2004.

[24] M. Tegmark, "An Icosahedron-based Method for Pixelizing the Celestial Sphere," *The Astrophysical Journal,* vol. 470, no. 2, pp. L81-L84, 1996.

[25] A. A. Eftekharian, C. Manion and M. I. Campbell, "Geometric Reasoning for Assembly Planning of Large Systems," in *Proceedings of the ASME 2013 International Design Engineering Technical Conference*, Portland, 2013.

[26] D. I. Agu, "Automated Analysis of Product Disassembly to Determine Environmental Impact," The University of Texas at Austin, Austin, 2009.

[27] Anomander, "Gear box with motor," 20 August 2012. [Online]. Available: https://grabcad.com/library/gear-box-with-motor.

[28] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing,* pp. 146-160, 1972.

[29] A. A. Eftekharian, R. Poladi and M. I. Campbell, "Evaluation and Search of Assembly Sequences in Large Systems," in *Proceedings of the ASME 2013*

International Design Engineering Technical Conference, Portland, 2013.